KATHOLIEKE
UNIVERSITEIT
LEUVEN

DEPARTEMENT TOEGEPASTE ECONOMISCHE WETENSCHAPPEN

RESEARCH REPORT 0033

IMPROVING THE REUSE POSSIBILITIES OF THE BEHAVIORAL ASPECTS OF OBJECT-ORIENTED DOMAIN MODELS

by
M. SNOECK
G. POELS

D/2000/2376/33

Improving the Reuse Possibilities of the Behavioral Aspects of Object-Oriented Domain Models

Monique Snoeck, Geert Poels

MIS Group,

Dept. Applied Economic Sciences, K.U.Leuven,
Naamsestraat 69,

3000 Leuven, Belgium

{monique.snoeck, geert.poels}@econ.kuleuven.ac.be

ABSTRACT. *Reuse of domain models is often limited to the reuse of the structural aspects of the domain (e.g. by means of generic data models). In object-oriented models, reuse of dynamic aspects is achieved by reusing the methods of domain classes. Because in the object-oriented approach any behavior is attached to a class, it is impossible to reuse behavior without at the same time reusing the class. In addition, because of the message passing paradigm, object interaction must be specified as a method attached to one class which is invoked by another class. In this way object interaction is hidden in the behavioral aspects of classes. This makes object interaction schemas difficult to reuse and customize. The focus of this paper is on improving the reuse of object-oriented domain models. This is achieved by centering the behavioral aspects around the concept of business events.*

This paper has been presented at the ER2000 Conference, 9-12 October, Salt Lake City, USA

1. INTRODUCTION

Domain modeling is an essential requirement capturing activity, prior to information systems modeling. As such, the main objective of any domain model is to be a vehicle for communication between system developers and business people, facilitating the mutual perception and understanding of important aspects of the business reality [17]. Although domain models are the particular representation of one or more aspects of a specific type of business (e.g. manufacturing, transportation, ...), the reuse of models from one domain to another is feasible (and supposedly also beneficial) when domains share a common knowledge structure. This principle of analogical reuse [13] has been supported by research contributions from various fields, including analysis patterns [7],[20], generic data models [8],[16], generic components [2], enumerative and faceted classification schemas [12], and automated pattern retrieval and synthesis [19]. Most of this work aims at facilitating the reuse of structural aspects of a domain (e.g. data models). Sometimes, in particular with respect to object-oriented modeling, it also concerns the reuse of functionality (e.g. object operations). In general however, the proposals that have been made do not concern the reuse of behavioral aspects related to the interaction of domain objects [14]. In object-oriented analysis, reuse is often centered around the reuse of class definitions. This type of reuse however, focuses on the reuse of design and code. Reuse at earlier stages of software development should focus on the reuse of analysis models. In object-oriented analysis there are typically at least three types of models, one for each view on the Universe of Discourse: a static model, an interaction model and a behavioral model. The goal of this research is to facilitate the reuse of the latter two types of models.

In this paper we present some research experiences with analogical reuse in the context of event-based domain modeling. In an event-based approach, the dynamic perspective of the domain is modeled by identifying the real-world events that are relevant to the universe of discourse. Domain objects are modeled in terms of their participation in real-world events (also called business events). In this way the dynamic perspective is modeled independently and at a high level of abstraction. This contrasts with the prevalent approach in OOA

that models the dynamic perspective through the concept of class-method, which is at a lower level of abstraction and subordinated to the concept of class. An important issue regarding the reuse of event-based domain models concerns the reuse of the participation of domain objects in real-world events, both in terms of the effect events have on domain objects as in terms of interaction between domain objects. The main focus of this paper is the improved reuse of such interaction aspects when an event-based approach is taken to conceptual domain modeling.

Our research concerns both the abstracting and customization of event-based domain models. We use an example throughout the paper to illustrate the possibilities and particularities of analogical reuse of domain object interaction schemas. In section 2 we present domain models for a library and a hotel administration along with a generic model that is a domain abstraction for these two analogous domains. The models in this section only represent structural aspects of the domain and take the form of UML class diagrams. Some issues regarding generalization and customization are illustrated and discussed. In section 3 the focus shifts to the modeling of behavioral aspects. First, behavior is added to the generic model following the rules prescribed by a formal method for object-oriented enterprise modeling [22],[23]. Next, the reuse of this behavior is illustrated and discussed. Section 4 then investigates the effect of required customization on the reuse of object interaction schemas. It is shown that an event-based approach to conceptual domain modeling improves the reuse possibilities of the object interaction schemas. More in particular, the effect of customization is shown to be less pervasive in an event-based interaction schema compared to a message passing interaction schema. Conclusions are presented in section 5.

2. A GENERIC DOMAIN MODEL FOR PRODUCT USAGE

Consider the following (simplified) domain descriptions for a library and a hotel administration:

"In the library we have a catalogue with titles and for each title the library has one or more copies. People can register to the library and become members. Members can borrow and return copies. Loans can be renewed. If a book is not on shelf, a reservation can be made for that title: the first copy that is returned to the library will then be put aside."

" A hotel offers a set of rooms that are categorized into room types. Customers make reservations for a particular room type. When the reservation is confirmed, a specific room is assigned for the customer's later stay."

The structural aspects of these domain descriptions are shown in Fig. 1 and Fig. 2 respectively.

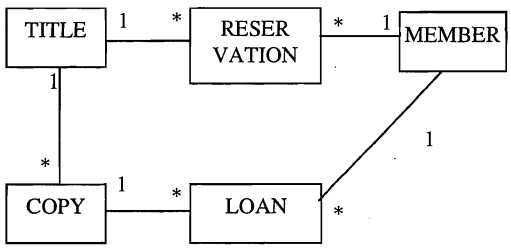

Fig. 1. A simple Library Domain Model

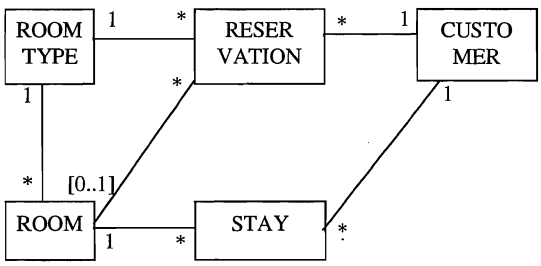

Fig. 2. A simple Hotel Administration Domain Model

As one can immediately notice, the class diagrams for the library and the hotel show a very similar structure. In both types of businesses products are categorized to product types. Customers can "use" a product during a certain

period of time, after which the product must be returned. Prior to this usage there may or may not be an "order" or reservation for the product's type.

The generic domain model for Product Usage is shown in Fig.3[1]. In this model, the association between USAGE_INTENTION and PRODUCT represents the allocation of products to reservations or orders. The association between USAGE_INTENTION and USAGE allows tracking how many of the effective usages are the consequence of a prior usage intention.

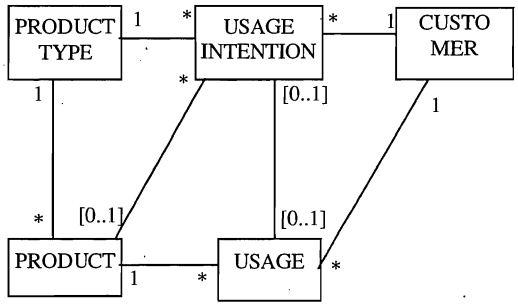

Fig. 3. A generic domain model for Product Usage

The generic model can also be extended to support multiple branches of one business. In the model of Fig. 4 we assume that product types are company-wide. However, the characteristics of a product type can be different from branch to branch: a double room in New York will have another (higher) price than a double room in Las Vegas. This requires the introduction of the class PRODUCT_TYPE_IN_BRANCH. Individual products are the materialization of such a PRODUCT_TYPE_IN_BRANCH and are as such located in one branch.

Although this generic domain model can be reused in many types of 'renting business', each domain will have its own particularities that must be taken

---

[1] In the classification framework of Lung and Urban [8] the domain abstraction for a library system and a hotel reservation system is called 'Object Allocation'. It is described as (p. 173) "an analogy for domains that allocate an object to another object (usually an agent). The allocated objects are returned after a period of time". Other example domains include car rental and airline reservation systems. Note that Lung and Urban do not propose generic models for their domain abstractions.

care of. Tailoring the generic structural model to the particularities of the own domain can be done by **adding** or **dropping** classes and/or associations, and by **considering additional business rules**. For example, in the library we will probably not be interested in keeping track of how many loans are the consequence of a reservation. As a result, the association between the RESERVATION class and the LOAN class has not been retained. In the case of the hotel administration, the decision whether or not to retain this association depends on the information needs of the specific company. For instance, the association must be retained if the hotel manager wishes to know for how many stays there was a prior reservation.

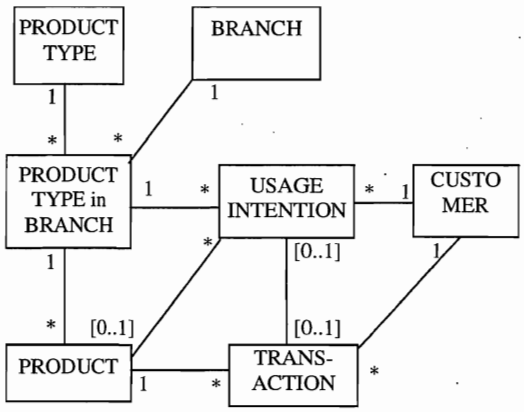

Fig. 4. An extended generic domain model for Product Usage

As another example, in a car rental domain model, which is another domain of the type Product Usage, it would also make sense to add an association between BRANCH and PRODUCT (i.e. a car) that records the current location of a car. This would allow customers to return the car to another branch than where it was rented. For example, it would allow customers to rent a car in the Brussels office and return it in the Paris office. In a library, the concept of PRODUCT_TYPE_IN_BRANCH makes less sense. It is sufficient to keep track of the location of each copy by directly linking COPY to LIBRARY (the branch) (Fig. 5).

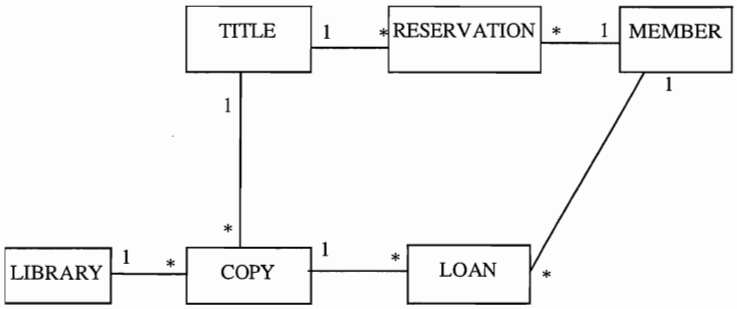

Fig. 5. Extended Library Domain Model

3. ADDING BEHAVIORAL ASPECTS

3.1. Motivation for an event-based approach

In the case of object-oriented conceptual modeling, domain requirements will be formulated in terms of business or enterprise object types, associations between these object types and the behavior of business object types. The definition of desired object behavior is an essential part in the specification process. On the one hand, we have to consider the behavior of individual objects. This type of behavior will be specified as methods and statecharts for object classes. On the other hand, objects have to collaborate and interact. Typical techniques for modeling object interaction aspects are interaction diagrams or sequence charts, and collaboration diagrams.

These techniques are based on the concept of message passing as interaction mechanism between objects. The main disadvantage of this concept is that in the context of domain modeling, message passing is too much implementation biased. We propose an alternative communication paradigm, namely, object interaction by means of joint involvement in business events. This type of interaction is modeled with an object-event table. Let us illustrate this with an example. In the context of a library, we can identify (among others) the two domain object types MEMBER and COPY. A relevant event type in this domain is the *borrowing* of a copy. This event affects both domain object types: it modifies the state of the copy and it modifies the state of the member. When

using message passing as interaction mechanism, two scenarios are possible. Either the member sends a message to the copy, or the copy sends a message to the member (see Fig. 6). If in addition LOAN is recognized as a domain object type as well, then the *borrow*-event will create loan objects. In this case, three objects are simultaneously involved in one event and should be notified of the occurrence of the *borrow*-event. With message passing, this leads to 9 possible interaction scenarios as depicted in Fig. 7. With each additional object type, the number of possible message passing scenarios further explodes. For example, if four objects have to synchronize on the occurrence of one event, we already have 64 possible message passing scenarios. Of course, from a systems design perspective, some scenarios can be considered more adequate than others. Domain modeling should however never be concerned with design aspects and business domain modelers should not be burdened with design considerations.

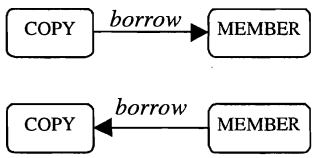

Fig. 6. Two possible scenarios for borrowing a copy

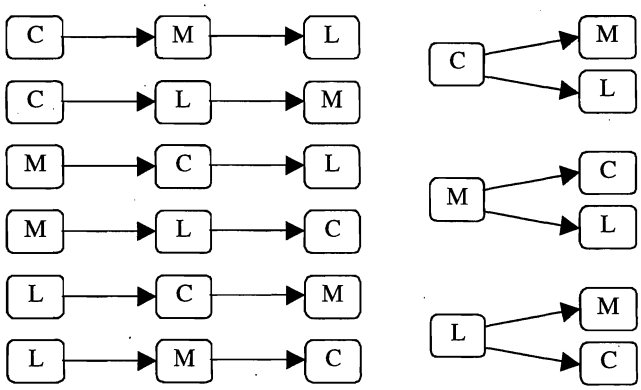

Fig. 7. Possible scenarios when three objects are involved in a single event (C = COPY, M = MEMBER, L = LOAN)

The alternative that we propose in this paper is to model only the essence of the interaction: some objects are affected by a given event, others are not. To model which objects are involved in which event types, we can use a very simple technique: the object-event table. Table 1. shows a possible object-event table for the library example. The table clearly shows that a *cr_member* event affects only the member object, that the acquisition of a copy only affects a copy, but that the borrowing and return of a copy affect a member, a copy and a loan object.

Table 1. Object-event table for the library

	MEMBER	COPY	LOAN
cr_member	×		
acquire		×	
borrow	×	×	×
return	×	×	×
...			

The use of the object-event table to model object interaction implies that the notion of **event** plays a central role. Some object-oriented analysis methods agree that events are a fundamental part of the structure of experience [4][6][21]. Events are atomic units of action: they represent things that happen in the real world. Without events nothing would happen: they are the way information and objects come into existence (creating events), the way information and objects are modified (modifying events) and disappear from our Universe of Discourse (ending events). As we are concerned with domain modeling, we will only consider ***business*** events (i.e. real world events) and, for example, not consider information systems events like keyboard and mouse actions. The concept of the object-event table allows to model interaction at a much higher level of abstraction than is the case with message passing. Moreover, the interaction pattern is independent of the number of objects involved in an event. At domain modeling level, we should not burden ourselves with event notification schemas. How exactly objects are notified of the occurrence of an event is a matter of implementation. When using object-oriented technology this will be done with messages, but when using other

technologies, both traditional and modern (e.g. distributed component technologies), (remote) procedure calls can do as well.

3.2. The generic behavioral schema for Product Usage

The specification of the behavioral aspects of the domain model consists of one object-event table and a set of lifecycle models, one for each of the domain classes. The object event table identifies the relevant event types for the Universe of Discourse and specifies the involvement of objects in events. In the object-event table (OET), events are not attached to a single domain class. One event can affect more than one object. In the object-event table, there is one column for each domain class and one row for each type of event relevant to the Universe of Discourse. A row-column intersection is marked with a 'C' when the event creates an object of the class, with an 'M' when it modifies the state of an object of the class and with an 'E' when it ends the life of an object of the class. A marked entry in a column means that, in an object-oriented implementation of the domain model, the domain class has to be equipped with a method to implement the effect of the event on the object. In this way the object-event table identifies the methods that have to be included in the class definition of domain objects.

For the (extended) generic domain model for Product Usage (Fig. 4) we identify the following event types:

create_customer, modify_customer, end_customer, create_branch, modify_branch, end_ branch, create_product_type, modify_product_type, end_product_type, allocate_product_ type_to_branch, modify_product_type_in_branch, end_product_type_in_branch, create_product, modify_product, end_product, cr_usage_intention, allocate_product, confirm_availability, cancel_usage_intention, start_usage, normal_return, abnormal_return, modify_conditions, invoice_usage, receive_payment, end_usage

The OET is represented in Table 2. A detailed discussion of the rules governing the construction of this OET is beyond the scope of this paper, but can

be found in [22], [23]. We merely note here that each marked entry identifies a possible place for information gathering. If for example, we wish to keep track of how many product types are offered in a branch, it makes sense to mark the entries BRANCH/*allocate_product_type_to_branch* and BRANCH/*end_product_type_in_branch.* Similarly, if within the class CUSTOMER we wish to keep track of the total amount of payments made by this customer (e.g. to identify "golden" customers, or to specify some discounting rules), we need to mark the entry CUSTOMER/ *receive_payment.* At implementation time, methods that are empty because no relevant business rule was identified, can be removed to increase efficiency.

Table 2. OET for the extended generic domain model for Product Usage.

	CUSTOMER	BRANCH	PRODUCT TYPE	PRODUCT TYPE IN BRANCH	PRODUCT	USAGE INTENTION	USAGE
Create_customer	C						
Modify_customer	M						
End_customer	E						
Create_branch		C					
Modify_branch		M					
End_branch		E					
Create_product_type			C				
Modify_product_type			M				
End_product_type			E				
Allocate_prod_type_to_branch		M	M	C			
Modify_prod_type_in_branch		M	M	M			
End_product_type_in_branch		M	M	E			
Create_product		M	M	M	C		
Modify_product		M	M	M	M		
End_product		M	M	M	E		
Cr_usage_intention	M	M	M	M		C	
Allocate_product	M	M	M	M	M	M	
Confirm_availability	M	M	M	M		M	
Cancel_usage_ intention	M	M	M	M		E	
Start_usage	M	M	M	M	M	E	C
Normal_return	M	M	M	M	M		M
Abnormal_return	M	M	M	M	M		M
Modify_conditions	M	M	M	M	M		M
Invoice_usage	M	M	M	M	M		M
Receive_payment	M	M	M	M	M		E
End_usage	M	M	M	M	M		E

Another behavioral aspect that is modeled concerns the specification of object lifecycle models. In the library for example, a copy should be returned before it can be borrowed again. With each class we will thus associate a lifecycle expression. The default lifecycle is that objects are first created (a choice between the C-entries), then modified an arbitrary number of times (an iteration of a choice between the M-entries) and finally come to an end (choice between the E-entries). In most object-oriented methods such lifecycles are represented using state charts. It is however also possible to represent such lifecycles as regular expressions, using a '+' to denote choice, a '.' to denote sequence and a '*' to denote iteration. From a mathematical and formal point of view, regular expressions are equivalent to state charts.

The lifecycle expression of a domain class should contain all events for which an entry has been marked in the corresponding column of the OET. In addition, the lifecycle expression should respect the type of the entries: events marked with a 'C' should appear as creating events, events marked with an 'M' should appear as modifying event types and events marked with an 'E' should terminate the life of the object[2]. For example the lifecycle expression for the class USAGE is represented in Fig. 8 as state chart and is specified as follows by means of a regular expression:

USAGE = *start_usage* . (*modify_conditions*)* . (*normal_return* + *abnormal_return*). *invoice_usage* . (*receive_payment* + *end_usage*)

---

[2] Additional rules that guarantee consistency between the object-relationship schema, the object-event table and the lifecycle expressions can be found in [22], [23].

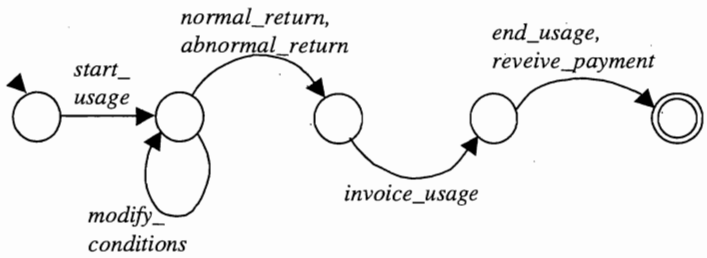

Fig. 8. State chart for USAGE

That is, after a usage has started, the conditions can be modified (e.g. postponing the return date) zero, once or more times. The product is then returned either in a normal state or in an abnormal state (e.g. crashed car). The usage is then invoiced and ends with the payment of the invoice or with the default *end_usage* event if the invoice gets never paid. The lifecycle for USAGE_INTENTION is:

USAGE_INTENTION = *create_usage_intenion . allocate_product . confirm . (cancel_ usage_intention + start_usage)*

When classes show some parallel behavior the '||' symbol is used to denote parallel composition in regular expressions, such as in the lifecycle of product:

PRODUCT = *create_product .*
*[(modify_product + allocate_product + invoice + receive_ payment + end_usage)**
|| (start_usage . (modify_conditions) . (normal_return + abnormal_return))*].*
end_product

That is, after a product has been created, its life is determined by two parallel threads. On the one hand there is the usage cycle and on the other hand there are a number of events that can occur randomly and independent from the usage cycle. The life of the product is terminated by the *end_product* event. Notice that constraints on event types such as *invoice* and *receive_payment* are already specified in the lifecycle of USAGE and need not be re-specified in the lifecycle of PRODUCT. The equivalent state chart is given in Fig. 9.

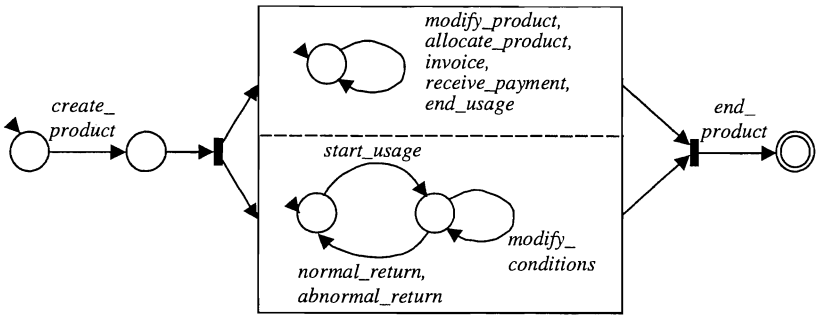

Fig. 9. State chart for PRODUCT.

The OET and object lifecycle models of the generic Product Usage model can be reused in the library, hotel administration and car rental domains. Again, some tailoring might be needed. For example, the way products are allocated to an intended transaction is similar in the hotel and car rental domains, but very different from the library domain. In a car rental and hotel business it is good practice to confirm the reservation to ensure that the requested product (i.e. a car or room) is available on the requested date. In a library however, such confirmation is not required: the member will simply receive the first copy that is returned and no firm assurance can be given on the data a copy will be available.

Reuse of the behavioral parts of the generic domain model is achieved in different ways. At the most abstract level behavior is reused by deciding which events to reuse and how. First, events can be reused as such by simply **renaming** them. For the car rental company, most event types can be reused by simply renaming them. For example, in the car-rental case *create_product_type* becomes *create_car_model, modify_product_ type* becomes *modify_car_model* and so on. Secondly, events can be **refined**. In the car-rental example, the *abnormal_return* can be split in two event types: *crash_car* and *total_loss.* Thirdly, events can be **added,** e.g. the event type *repair* can be added to allow putting a car back in circulation after a crash. Finally, events can be **dropped**. For example in a library there is no need to allocate free books to reservations. Hence, the event types *allocate_ product* and *confirm_availability* are dropped.

At a more detailed level of specification individual class behavior is reused by refining the life cycle expressions of object types according to the modified event type definitions. For example, the life cycle of CAR becomes more complex as we want to specify that after a total loss a car can never be rented again and that after a crash, the car needs repairing.

CAR = *buy_car* .
[(*modify_car_details* + *allocate_car* + *invoice* + *receive_ payment* + *end_rental*)*
|| (*rent* . (*change_return_date*)* . (*normal _return* + *crash_car.repair*))*
.(1 + (*rent* . (*change_return_date*)* .*total_loss*)]
. *end_car*

In this lifecycle the '1' stands for the empty event. The lifecycle thus specifies that after an arbitrary number of rent-cycles either nothing special happens or we have one final rent cycle that ends with the total loss of the car. The equivalent state chart is given in Fig. 10.

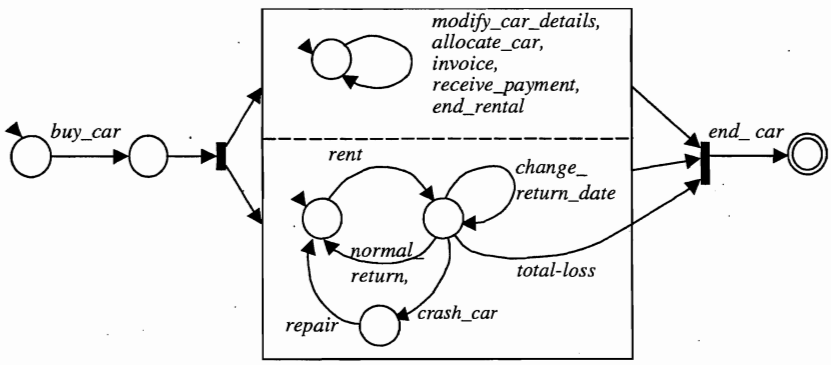

Fig. 10. State chart for CAR.

In the library example, the life cycle of COPY is refined to specify that after a copy has been lost it can never be borrowed again:

COPY = *classify_copy* .
[(*modify_copy_details* + *fine* + *receive_ payment* + *end_loan*)*
|| (*borrow* . (*renew*)* . *return*)*.(1 + (*borrow* . (*renew*)* .*lose*))]
. *end_copy*

4. IMPROVED REUSE OF THE OBJECT INTERACTION SCHEMA

The most important implication of the use of the object-event table resides in the modeling of object interaction. In the approach proposed in the previous section, it is assumed that events are broadcasted to objects. This means that when an event occurs and is accepted, all corresponding methods in the involved objects will be executed simultaneously provided each involved object is in a state where this event is acceptable. This way of communication is similar to communication as defined in the process algebras CSP [9] and ACP [1] and has been formalized in [5], [23]. Message passing is more similar to the CCS process algebra [15]. There exist various mechanisms for the implementation of such synchronous execution of methods. For the purpose of analyzing the effects on reuse, we will assume that there is an event handling mechanism that filters the incoming events by checking all the constraints this event must satisfy. If all constraints are satisfied, the event is broadcasted to the participating objects; if not it is rejected. In either case the invoking class is notified accordingly of the rejection, acceptation, and successful or unsuccessful execution of the event. This concept is exemplified in Fig. 11. for part of the generic schema of Fig. 3. For each type of business events, the event handling layer contains one class that is responsible for handling events of that type. This class will first check the validity of the event and, if appropriate, broadcast the event to all involved objects by means of the method 'run'.

In a conventional object-oriented approach, object interaction is achieved by having objects send messages to each other. This is documented by means of collaboration diagrams. Because of the absence of the broadcasting paradigm, events must be routed through the system in such a way that all concerned objects are notified of the event. As there is no generally accepted schema, the routing schema must be designed for each type of event individually. An additional problem is the identification of the object where the routing will start. In most examples given in object-oriented analysis textbooks, the business events are initially triggered by some information system event. For example, in an ATM system, the *withdraw_amount* business event is triggered by the information system event *insert_card.* Such interactions can be represented by including information system objects such as user interface objects in the

collaboration diagram. From a domain modeling perspective, we would prefer object interaction to be independent from information system services. For example, the business event *withdraw_amount* can also be triggered by other information system services such as the counter application. In order to represent interaction independent from information system services, in the collaboration diagram below, a dummy class is included that represents the business event invocation. The routing of the event starts in that class and is then routed through the domain model in such a way that all domain classes affected by this type of event are notified. Fig. 12 shows possible interaction schemas for the *cr_product* and the *allocate_product* event types. Notice that because the *allocate_product* event type affects four different domain classes for this event type there are 64 possible routing schemas that allow to notify all 4 objects of the occurrence of an *allocate_product* event (see discussion in section 3.1).

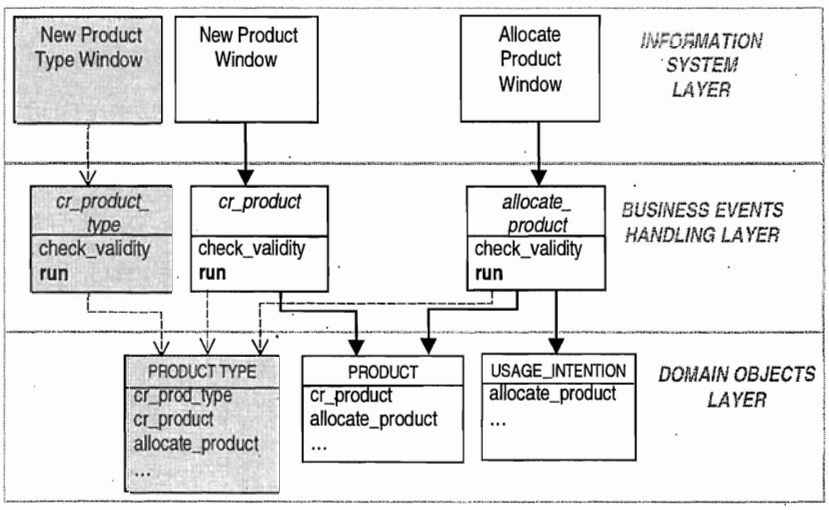

Fig. 11. Part of an event broadcasting schema for the generic schema

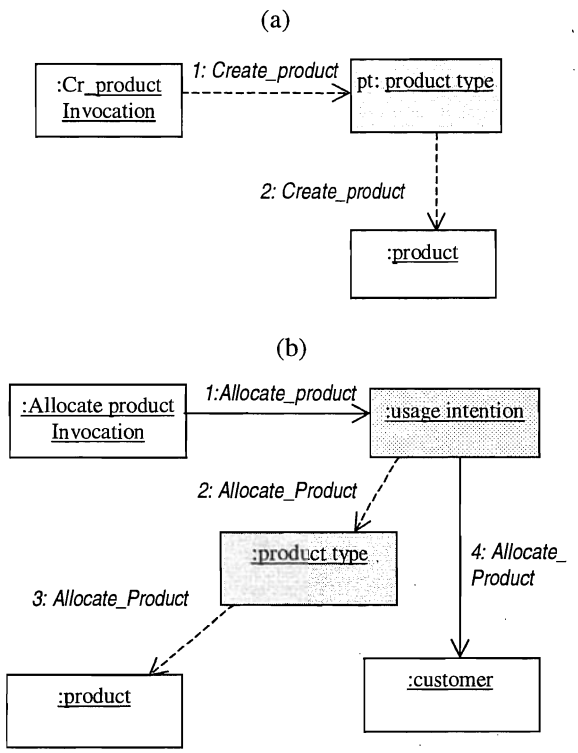

Fig. 12. Collaboration diagrams for the generic schema of Fig. 3.

When the generic schema is customized, object types and event types can be added to, refined or dropped from the generic schema. Such changes turn out to be less pervasive for the broadcasting paradigm than for the message passing paradigm. Let us assume for example that the generic schema is reused for a Small Car rental Company, were the object class PRODUCT_TYPE is not required: in this Small Car rental Company reservations are made directly for individual cars. For the broadcasting schema this means that except for the removal of the product type domain class, all modifications are localized in the event handling layer. The required modifications are shown as shaded areas in Fig. 11. The effect on the collaboration diagrams is more pervasive: the whole interaction schema must be redesigned (see shaded areas in Fig. 12). The modification of the interaction schema even requires modifications in other domain classes. For example, in Fig. 12 (b), the removal of the PRODUCT_ TYPE domain class, requires a modification of the USAGE_INTENTION

domain class as this class must now propagate the *allocate_product* event directly to PRODUCT rather than to PRODUCT_TYPE (i.e. send a message to PRODUCT instead of PRODUCT_TYPE). Similarly, adding a domain class has a more limited effect on the broadcasting schema compared to the classical approach. Let us for example add the BRANCH and PRODUCT_TYPE_IN_BRANCH domain classes such as to obtain the generic schema of Fig. 4. In the broadcasting schema the effect for the existing classes is limited to the event handling layer as exemplified in Fig. 13. For the conventional interaction schema documented with collaboration diagrams, the effect is again more pervasive. Depending on the new routing schema for events, the modifications also propagate to one or more existing domain classes. Fig. 14 shows an example of a new collaboration diagram for the *allocate_product* event type, which requires a modification of the PRODUCT domain class.

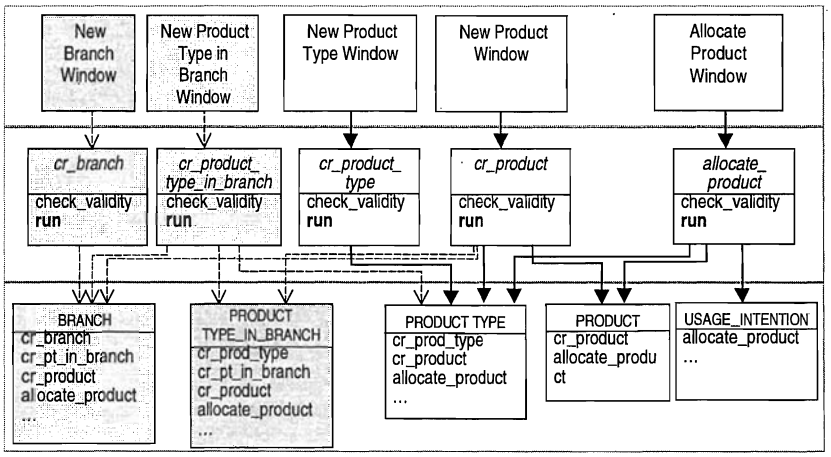

Fig. 13. Effect of adding the domain classes BRANCH and PRODUCT_ TYPE_ IN_BRANCH to the event broadcasting schema for the generic schema

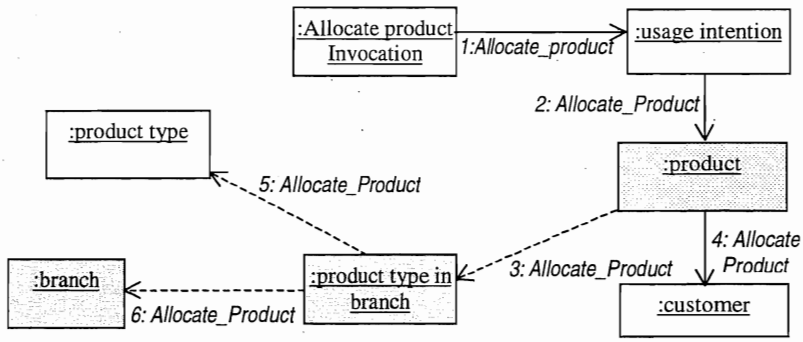

Fig. 14. Modified collaboration diagram for *allocate_product*

5. CONCLUSIONS

In this paper we considered various issues related to the generalization of 'analogous' domain models and the customization of the resulting generic domain models. The example indicates that the reuse of both structural and behavioral aspects of domain models is certainly possible. The most difficult part to reuse is the object interaction aspect [14]. It is a well-known fact that if no special effort is taken to minimize the number of collaborations, interaction diagrams quickly take an aspect of spaghetti [25]. In addition, when interaction between domain object classes is not centered around the concept of business events, it is hidden in the methods of object classes. It then is very difficult to evaluate the impact of customization on the interaction schema. Depending on the chosen scenario for event propagation, one or more domain classes will require some adaptation to implement the modified scenario. The use of the broadcasting paradigm simplifies the reuse of object interaction aspects substantially. Moreover, the required modifications are more systematic and hence easier to trace, and the domain classes are better isolated from modifications such as the addition or removal of domain classes.

It is also important to notice that the broadcasting paradigm allows a system with a layered structure as shown in Fig. 11: information system objects and domain objects are kept in separate layers, with the event handling layer playing the role of "event broker" [24]. These layers also reflect an appropri-

ate separation of concerns, namely the separation of domain knowledge and business rules from information system support. In addition, as the services of objects in a layer are only used by objects of an upper layer, modifications in the upper layer do not propagate to lower layers. This not only makes customization of generic models easier, but also facilitates system maintenance.

Because of the modeling of business events as first-class citizens in the domain model, the reuse of behavior can be considered by looking only at the event types. In a way, the choice of columns and rows in the OET to reuse can be done independently of each other. In a classical approach you would choose the classes in the structural model and hope that they contain the required behavior.

It must be noticed that the reuse of domain models cannot be considered on its own. Domain models must be seen as reusable software requirements. Defining a domain model is part of the requirements engineering step in the development of an information system: all business rules described in the domain model have to be supported by the information system. Methods such as JSD [10], OO-SSADM [21], Syntropy [4], Catalysis [6], and MERODE [23][22] even explicitly define domain modeling as a separate step in the development process. Jacobson [11] assumes the existence of a domain model that serves as a basis to identify entity objects. As such object interaction schemas, which capture a major part of the business rules governing a domain, can be considered as reusable specifications. The event-based approach to conceptual domain modeling assumed in this paper greatly enhances their reuse possibilities.

The generic domain model presented in this paper models domains of the type Product Usage, included as the domain abstraction Object Allocation in the classification framework of Lung and Urban [12]. We have worked on, and continue to work on, generic models for other domain abstractions. A related topic of research is the definition of distance measures for event-based, object-oriented domain models [18]. Such measures, similar in concept to the similarity measures for components of Castano et al. [3], allow to quantify and evaluate the conceptual distance between domains. This information can for

instance be used to decide whether analogical reuse is feasible, i.e. whether it is worth reusing from a generic domain model.

Acknowledgements

Geert Poels is a Postdoctoral Fellow of the Fund for Scientific Research - Flanders (Belgium)(F.W.O) and wishes to acknowledge the financial support of the Fund for Scientific Research.

6. REFERENCES

[1] Baeten, J.C.M.: Procesalgebra: een formalisme voor parallelle, communicerende processen. Kluwer programmatuurkunde, Kluwer Deventer (1986)

[2] Castano, S., De Antonellis, V.: The F³ Reuse Environment for Requirements Engineering. ACM SIGSOFT Software Eng. Notes 19 (1994) 62-65

[3] Castano, S., De Antonellis, V., Pernici, B.: Building Reusable Components in the Public Administration Domain. In: Proc. ACM SIGSOFT Symposium Software Reusability (SSR'95). Seattle (1995) 81-87

[4] Cook, S., Daniels, J.: Designing object systems: object-oriented modeling with Syntropy. Prentice Hall (1994)

[5] Dedene, G., Snoeck, M.: Formal deadlock elimination in an object oriented conceptual schema. Data and Knowledge Eng. 15 (1995) 1-30

[6] D'Souza, D.F., Wills, A.C.: Objects, components, and frameworks with UML: the catalysis approach. Addison-Wesley (1998)

[7] Fowler, M.: Analysis Patterns: Reusable Object Models. Addison-Wesley (1997)

[8] Hay, D.C.: Data Model Patterns: Conventions of Thought. Dorset House Publishers, New York (1996) .

[9] Hoare, C. A. R.: Communicating Sequential Processes. Prentice-Hall (1985)

[10] Jackson, M.A.: System Development. Prentice Hall (1983)

[11] Jacobson, I. et al.: Object-Oriented Software Engineering, A use Case Driven Approach. Addison-Wesley (1992)

[12] Lung, C.-H., Urban, J.E.: An Approach to the Classification of Domain Models in Support of Analogical Reuse. In: Proc. ACM SIGSOFT Symposium Software Reusability (SSR'95). Seattle (1995) 169-178

[13] Maiden, N.A., Sutcliffe, A.G.: Exploiting Reusable Specifications Through Analogy. Communications of the ACM 35 (1992) 55-64

[14] Mili, H., Mili, F., Mili, A.: Reusing Software: Issues and Research Directions. IEEE Trans. Software Eng. 21 (1995) 528-561

[15] Milner R.: A calculus of communicating systems. Springer Berlin, Lecture Notes in Computer Science (1980)

[16] Mineau, G.W., Godin, R.: Automatic structuring of knowledge bases by conceptual clustering. IEEE Trans. Data and Knowledge Eng. 7 (1995) 824-829

[17] Nellborn, C.: Business and Systems Development: Opportunities for an Integrated Way-of-Working. In: Nilsson, A.G., Tolis, C., Nellborn, C. (eds.): Perspectives on Domain modeling: understanding and Changing Organisations. Springer Verlag, Berlin (1999)

[18] Poels, G., Viaene, S., Dedene, G.: Distance Measures for Information System Reengineering. In: Proc. 12th Int'l Conf. Advanced Systems Eng. (CAiSE*00). Stockholm (2000) 387-400

[19] Purao, S., Storey, V.C.: Intelligent Support for Retrieval and Synthesis of Patterns for Object-Oriented Design. In: Proc. 16th Int'l Conf. Conceptual Modeling (ER'97). Los Angeles (1997) 30-42

[20] Robertson, S.: Mastering the Requirements Process. Addison-Wesley (1999)

[21] Robinson, K., Berrisford, G.: Object-oriented SSADM. Prentice Hall (1994)

[22] Snoeck, M., Dedene, G.: Existence Dependency: the key to semantic integrity between structural and behavioral aspects of object types. IEEE Trans. Software Eng. 24 (1998) 233-251

[23] Snoeck, M., Dedene, G., Verhelst, M., Depuydt, A.: Object-oriented Enterprise Modeling with MERODE. University Press, Leuven (1999)

[24] Snoeck, M., Poelmans, S., Dedene, G., A Layered Software Specification Architecture. In: Proc. 19th Int'l Conf. Conceptual Modeling (ER2000). Salt Lake City (2000)

[25] Wirfs-Brock, R., Johnson, R.E.: Surveying current research in OO design. Communications of the ACM 33 (1990) 105-124